Guide to **Architecture**

Edited by

Steve Rabin

CRC Press
Taylor & Francis Group

Game AI Pro 360

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Game AI Pro 360
Guide to Architecture

Edited by
Steve Rabin

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2020 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper

International Standard Book Number-13: 978-0-367-15104-1 (paperback)
International Standard Book Number-13: 978-0-367-15107-2 (hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

About the Editor ix

About the Contributors xi

Introduction xix

1 Behavior Selection Algorithms: An Overview 1
Michael Dawe, Steve Gargolinski, Luke Dicken, Troy Humphreys, and Dave Mark

2 Structural Architecture—Common Tricks of the Trade 15
Kevin Dill

3 The Behavior Tree Starter Kit 27
Alex J. Champandard and Philip Dunstan

4 Real-World Behavior Trees in Script 47
Michael Dawe

5 Simulating Behavior Trees: A Behavior Tree/Planner Hybrid Approach 53
Daniel Hilburn

6 An Introduction to Utility Theory 67
David "Rez" Graham

7 Building Utility Decisions into Your Existing Behavior Tree 81
Bill Merrill

8 Reactivity and Deliberation in Decision-Making Systems 91
Carle Côté

9 Exploring HTN Planners through Example 103
Troy Humphreys

10 Hierarchical Plan-Space Planning for Multi-unit Combat Maneuvers 123
William van der Sterren

11 Phenomenal AI Level-of-Detail Control with the LOD Trader 139
Ben Sunshine-Hill

12 Runtime Compiled C++ for Rapid AI Development 155
Doug Binks, Matthew Jack, and Will Wilson

13 Plumbing the Forbidden Depths: Scripting and AI 173
Mike Lewis

14 Possibility Maps for Opportunistic AI and Believable Worlds 193
John Manslow

15 Production Rules Implementation in *1849* 201
Robert Zubek

16 Production Systems: New Techniques in AAA Games 211
Andrea Schiel

17 Building a Risk-Free Environment to Enhance Prototyping: Hinted-Execution Behavior Trees 221
Sergio Ocio Barriales

18 Smart Zones to Create the Ambience of Life 241
Etienne de Sevin, Caroline Chopinaud, and Clodéric Mars

19 Separation of Concerns Architecture for AI and Animation 253
Bobby Anguelov

20 Optimizing Practical Planning for Game AI 269
Éric Jacopin

21 Modular AI 281
Kevin Dill and Christopher Dragert

22 Overcoming Pitfalls in Behavior Tree Design 309
Anthony Francis

23 From Behavior to Animation: A Reactive AI Architecture for Networked First-Person Shooter Games 321
Sumeet Jakatdar

24 A Character Decision-Making System for *FINAL FANTASY XV* by Combining Behavior Trees and State Machines 339
Youichiro Miyake, Youji Shirakami, Kazuya Shimokawa, Kousuke Namiki, Tomoki Komatsu, Joudan Tatsuhiro, Prasert Prasertvithyakarn, and Takanori Yokoyama

25 A Reusable, Light-Weight Finite-State Machine 353
David "Rez" Graham

26 Choosing Effective Utility-Based Considerations 361
Mike Lewis

27 Combining Scripted Behavior with Game Tree Search for Stronger, More Robust Game AI 373
Nicolas A. Barriga, Marius Stanescu, and Michael Buro

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

About the Editor

Steve Rabin has been a key figure in the game AI community for more than a decade and is currently a principal software engineer at Nintendo Technology Development. After initially working as an AI engineer at several Seattle startups, he managed and edited seven game AI books in the "Game AI Pro" series and the "AI Game Programming Wisdom" series. He also edited the book *Introduction to Game Development* and has more than two dozen articles published in the "Game Programming Gems" series. He has been an invited keynote speaker at several AI conferences, founded the AI Game Programmers Guild in 2008, and founded the GDC AI Summit where he has been a summit adviser since 2009. Steve is a principal lecturer at the DigiPen Institute of Technology, Redmond, Washington, where he has taught game AI since 2006. He earned a BS in computer engineering and an MS in computer science, both at the University of Washington, Seattle, Washington.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

About the Contributors

Bobby Anguelov is a South African expat currently working as a senior programmer at Ubisoft, Montreal, Canada, primarily in the field of animation. Prior to joining Ubisoft, he worked as an AI/animation programmer at IO-Interactive. Outside of the game industry, he has worked in a variety of roles ranging from computer graphics lecturer to enterprise software consultant. Bobby holds an MS in computer science from the University of Pretoria, South Africa, with a focus on computational intelligence and computer graphics.

Nicolas A. Barriga is a PhD candidate at the University of Alberta, Edmonton, Canada. He earned BSc (engineering) and MSc degrees in informatics engineering at Universidad Técnica Federico Santa María, Valparaíso, Chile. After a few years working as a software engineer for Gemini and ALMA astronomical observatories, he came back to graduate school and is currently working on state and action abstraction mechanisms for RTS games.

Doug Binks makes games at Enkisoftware Limited, having recently left his position as technical lead of Games Architecture Initiative at Intel. Prior to joining Intel in 2008 he worked in the games industry in roles ranging from lead programmer, head of studio at Strangelite, and R&D development manager at Crytek. Despite an early interest in games development, Doug careered sideways into a doctorate in physics at Oxford University, and undertook two postdoctoral posts as an academic researcher in experimental non-linear pattern formation, specializing in fluid mechanics. His earliest memories are of programming games in assembly on the ZX81.

Michael Buro is a professor in the computing science department at the University of Alberta, Edmonton, Canada. He received his PhD in 1994 for his work on *Logistello*—an Othello program that defeated the reigning human World champion 6-0. His current research interests include heuristic search, pathfinding, abstraction, state inference, and opponent modeling applied to video games and card games. In these areas, Michael and

his students have made numerous contributions, culminating in developing fast geometric pathfinding algorithms and creating the World's best Skat playing program and one of the strongest *StarCraft: Brood War* bots.

Alex Champandard is the founder of AiGameDev.com, the largest online hub for artificial intelligence in games. He has worked in industry as a senior AI programmer for many years, most notably for Rockstar Games where he also worked on the animation technology of *Max Payne 3*. He regularly consults with leading studios in Europe, most notably at Guerrilla Games on the multiplayer bots for *KillZone 2* & *3*. Alex is also the event director for the Game/AI Conference, the largest independent event dedicated to AI in games.

Caroline Chopinaud received her PhD in computer science at the Pierre and Marie Curie University, Paris, France, in 2007, specializing in artificial intelligence. She joined MASA Group in 2008 as a developer of MASA LIFE, working on agent communications and the modeling of security and military behaviors. Caroline has since become head of R&D at MASA, in charge of innovation and the R&D team. She is mainly involved in research activities, in particular scientific monitoring and project building. She handles relations between MASA and academia. She published scientific papers in international conferences in the field of multiagent systems (ECAI, IAT, PAAMS, etc.).

Carle Côté has been a senior AI programmer at Eidos Montreal since 2009 and currently leads the AI development on the next *Thief* game. In 2012, he received his PhD in electrical engineering applied to AI and robotics from Sherbrooke University in Canada. His focus is mainly on decision-making systems and cognitive AI.

Michael Dawe has been solving AI and gameplay problems in games since 2007, working on games such as *Kingdoms of Amalur: Reckoning*, *Dance Central Spotlight*, and *Grand Theft Auto 5*. He is a frequent speaker for the AI Summit at the Game Developers Conference, is a founding member of the AI Game Programmers Guild, and has written previously for the Game Programming Gems and Game AI Pro series. Michael holds an MS in computer science from DigiPen Institute of Technology, Redmond, Washington, as well as a BSc in computer science and philosophy from Rensselaer Polytechnic Institute, Troy, New York.

Etienne de Sevin earned his PhD in artificial intelligence at École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, in 2006. During his research in several renowned labs, he designed, implemented, and evaluated cognitive architectures and real-time decisionmaking processes for autonomous nonplayer characters in order to populate virtual environments such as virtual cities or video games. He published several scientific papers in international conferences (AAMAS, CGI, CASA, etc.). He joined MASA Group as a research engineer in the R&D team in 2012.

Luke Dicken is the founder of Robot Overlord Games, and a researcher with the Strathclyde Artificial Intelligence and Games group at the University of Strathclyde in the United Kingdom. He contributes to AltDevBlogADay and is a principal organizer for the AltDev Conference family. Luke has been passionate about artificial intelligence since

playing *Creatures* as a teenager, and pursued it in college, first through several degrees in traditional AI before specializing in AI for games as part of a PhD he is still (occasionally) pursuing. Luke is a member of the AI Game Programmers Guild, on the board of directors for IGDA Scotland, and recently took over as chair of the IGDA's Special Interest Group on AI.

Kevin Dill is a member of the senior technical staff at Lockheed Martin Rotary & Missions Systems and the chief architect of the Game AI Architecture. He is a veteran of the game and military simulation industries with more than 15 years' experience, and has worked on AI for everything from games (including several major hits, such as *Red Dead Redemption*, *Iron Man*, and *Zoo Tycoon 2*) to military training to emotive avatars. His professional interests include improved techniques for behavior specification, tactical and strategic AI, spatial reasoning, and believable characters. He was the technical editor for *Introduction to Game AI* and *Behavioral Mathematics for Game AI*, and a section editor for "AI Game Programming Wisdom 4" and the "Game AI Pro" series. He is a prolific author and speaker, and has taught at Harvard University, Cambridge, Massachusetts, Boston University, Boston, Massachusetts, and Northeastern University, Boston, Massachusetts.

Christopher Dragert received his PhD in computer science from McGill University, Montréal, Canada. His research focused on the application of software engineering techniques to the development process of game AI, with work being published at top academic conferences, including AIIDE, FDG, FSE, and GAS. Recently, he spoke at GDC 2016 and authored a chapter in the recent book *Computer Games and Software Engineering*. Chris is currently employed at Ubisoft Toronto as an AI programmer on *Watch Dogs 2*.

Philip Dunstan, as a senior AI R&D engineer at AiGameDev.com, prototypes cutting-edge solutions to the artificial intelligence challenges found in today's games. In addition, Philip has 6 years of development experience within Electronic Arts' EATech Central Technology Group. As a specialist in physics simulation, core technology, and console performance, he worked on several of EA's biggest franchises including *FIFA*, *Need for Speed*, *Battlefield*, and *Harry Potter*.

Anthony Francis, by day, studies human and other minds to create intelligent machines and emotional robots; by night, he writes urban fantasy and draws comic books. His work explores deep learning for robotic control at Google Research. He earned his PhD at Georgia Institute of Technology, Atlanta, Georgia, studying contextual memory for information retrieval, but also worked on emotional long-term memory for robot pets, a project documented on the Google Research blog as "Maybe your computer just needs a hug." Anthony is best known for his "Skindancer" series of urban fantasy novels featuring magical tattoo artist Dakota Frost, including the award-winning *Frost Moon* and its sequels *Blood Rock* and *Liquid Fire*. Anthony lives in San Jose with his wife and cats, but his heart will always belong in Atlanta. You can follow Anthony online at http://www.dresan.com/

Steve Gargolinski has been working on games professionally since 2003, spending time at Blue Fang Games, Rockstar New England, and 38 Studios. Steve has a strong technical

background, and enjoys thinking, writing, and speaking about game AI, programming, and the development process. He has presented at conferences such as the Game Developers Conference (GDC) and the AI and Interactive Digital Entertainment Conference (AIIDE), and has been interviewed by *The Independent* and *Gamasutra* for his work in gaming AI. While not programming computers Steve enjoys nonfiction, cooking, hockey, and walking in the woods.

David "Rez" Graham is a senior AI programmer currently working independently on a self-funded unannounced project. Before that, he was the director of game programming at the Academy of Art University's School of Game Development, San Francisco, California, where he built the entire game programming curriculum from scratch. He has worked in the game industry as an engineer since 2005, spending most of that time working on various kinds of AI, from platformer enemy AI to full simulation games. Most recently, he was the lead AI programmer on *The Sims 4* at Maxis. Before that, he worked at PlayFirst, Slipgate Ironworks, Planet Moon Studios, and Super-Ego Games. He is the coauthor of *Game Coding Complete* (4th edition), has two articles in *Game AI Pro*, and regularly speaks at the Game Developers Conference. Rez spends his free time acting and running tabletop RPGs.

Daniel Hilburn has been making video games since 2007. He has worked on several console games including *Kinect Star Wars*™, *Ghostbusters: The Video Game*™, and *DefJam's Rapstar*™. He currently works in Irving, Texas, at Terminal Reality, Inc.

Troy Humphreys has been involved in game mechanics and AI since 2005. Since then, he has worked on the games *The Bourne Conspiracy*, *Transformers: War for Cybertron*, and *Transformers: Fall of Cybertron*. He currently works as a senior programmer at High Moon Studios, where he helps lead the studio's AI development. Prior to working on games, he taught game development as an Associate Course Director at Full Sail, where he still serves as an adviser.

Matthew Jack founded Moon Collider (www.mooncollider.com) in 2010, where he consults on AI for companies in the US and Europe and builds bespoke AI systems. He specializes in *CryEngine 3* and *Recast/Detour*. He developed AI at Crytek for many years in a senior R&D role, including work on *Crysis* and *Crysis 2*. He has since worked for Microsoft and AiGameDev.com, and consulted for games and serious games companies. Clients include Xaviant LLC and Enodo, with products delivered to companies such as BMW. He has written for *Games Programming Gems* and presented at the GDC, Paris Game AI Conference, Develop and at Google.

Éric Jacopin is a professor at the French Military Academy of Saint-Cyr, Guer, France where he headed the Computer Science Research Laboratory from 1998 to 2012; this includes teaching Turing machines, war games and project management, and the management of international internships for computer science cadets. His research has been in the area of AI Planning for the past 25 years, not only from the viewpoint of artificial intelligence but also from everyday life and gaming perspectives. He received his PhD (1993) and his habilitation to direct research (1999), both from the Pierre and Marie Curie University (Paris VI), Paris, France.

Sumeet Jakatdar is the lead AI engineer at Treyarch/Activision Blizzard, Santa Monica, California, and has been working there for more than 9 years. He has been involved in developing AI for some of the Activision's most popular first-person shooter games. During his years as an AI developer, he has worked on gameplay programming, level design, behavior, animation systems, and networking. Before joining the industry in 2007, he received MS degree in computer science from University of Southern California, Los Angeles, California.

Tomoki Komatsu graduated from Tohoku University, Sendai, Japan, and completed the master's degree in computer science. Then he started his career as an AI Engineer at SQUARE ENIX in 2014. He developed the Monster AI and supported the development of the tools for *FINAL FANTASY XV*.

Mike Lewis entered the game industry as a programmer in early 2002, and has spent most of the intervening years focusing on game AI and surrounding technologies. He has lectured at the Game Developers Conference and published articles in previous volumes of *Game AI Pro*. Currently, Mike calls ArenaNet, LLC, home, where he tirelessly schemes to bring better AI to the world of massively multiplayer online gaming.

John Manslow started writing games on his Vic-20 as a teenager and gradually became more and more interested in smart AI. Having completed a degree and then a PhD in the subject, he joined Codemasters as the AI specialist in their R&D team, where he worked on several projects to bring the next generation AI to Codemasters' games. Since then, John has worked for several companies outside the industry but has remained focused on AI and statistical analytics.

Dave Mark is the president and lead designer of Intrinsic Algorithm, an independent game development studio in Omaha, Nebraska. He does consulting on AI, game design, and mathematical modeling for clients ranging from small indie game studios to AAA companies including EA, Sony Online Entertainment, and ArenaNet. Dave is the author of the book *Behavioral Mathematics for Game AI* and is a contributor to the *AI Game Programming Wisdom* and *Game Programming Gems* book series from Charles River Media and the first Game AI Pro book from CRC Press. He has also spoken at numerous game conferences and universities around the world on the subjects of AI, game theory, and psychology. He is a founding member of the AI Game Programmers Guild and has been a coadvisor of the Game Developers Conference AI Summits. Dave continues to further his education by attending the University of Life. He has no plans to graduate anytime soon.

Clodéric Mars has tried to make simulated characters behave "autonomously *and* as they are told to" for more than 6 years. At Golaem, he worked on a navigation engine used, for example, in a train passengers' simulation and a crowd simulation tool for animation and vfx. Now leading the developments of MASA LIFE, he is dedicated to make behavior authoring easy, fun, and accessible to game designers and field experts. Clodéric has a master's degree in computer science, with a specialization in AI. He spoke at the Paris Game/AI Conference 2011 and at the Game Developers Conference (GDC) 2014 AI Summit.

Bill Merrill is the AI lead at Turtle Rock Studios working hard on an unannounced project, having previously worked as AI lead and senior generalist at Double Helix Games, shipping cross-platform game projects including *Dirty Harry*, *Silent Hill: Homecoming*, *Front Mission: Evolved*, and various tools and demos. He currently splits his time between technology and toddlers.

Youichiro Miyake is the lead AI researcher at Square Enix, working as the leader of the AI unit for the next-generation game engine Luminous Studio. He is the chairman of the IGDA JAPAN SIG-AI and a board member of DiGRA JAPAN. He has been developing and researching game AI since 2004. He developed the technical design of AI for the following game titles: *Chromehounds* (2006, Xbox 360), *Demon's Souls* (2009, PlayStation 3), and *Armored Core V* (2012, Xbox 360, PlayStation 3), developed by FromSoftware. At Square Enix, he was engaged in the AI development of *FINAL FANTASY XIV*: *A Realm Reborn*. At present, he is developing AI in *FINAL FANTASY XV* as the lead AI technical architect. He has published papers and books about game AI technologies and has given many lectures at universities and conferences. He was a keynote speaker of GAMEON ASIA 2012 and a game AI course speaker in SIGGRAPH ASIA 2015. His paper "Current Status of Applying Artificial Intelligence in Digital Games" will be published in the *Handbook of Digital Games and Entertainment Technologies* by Springer.

Kousuke Namiki joined FROM SOFTWARE in 2009 and worked on *STEEL BATTALION* and *Monster Hunter Diary: Poka Poka Airou Village*. He has been with SQUARE ENIX since 2012 as an AI programmer developing Luminous Studio. Now he is engaged in the development of enemy character AI for *FINAL FANTASY XV*.

Sergio Ocio Barriales has been working in the game industry since 2005. He received his PhD in 2010 from the University of Oviedo, Asturias, Spain, with his thesis about hinted-execution behavior trees. He has worked on the AI for numerous major titles, such as *Driver San Francisco*, *Splinter Cell: Blacklist*, *DOOM*, and *Watch_Dogs 2*. He joined the team at Hangar 13 as a lead AI engineer in 2016, where he continues pushing character AI forward.

Prasert Prasertvithyakarn is the lead game designer for *FINAL FANTASY XV* and is responsible for the game's AI main player characters. He was the leader of the enemy team in *MAJIN AND THE FORSAKEN KINGDOM* (2010, Xbox360, PlayStations3) at GAME REPUBLIC INC. before joining SQUARE ENIX in 2010.

Andrea Schiel has more than 18 years of experience developing AAA games and has worked on most of the major EA Sports™ titles. A recognized specialist in AI, she leads the AI special interest group at EA™ and mentors other engineers in this field. Andrea has shipped over 25 titles on all major platforms and is currently working for BioWare Montreal™ on the *Mass Effect*™ franchise.

Kazuya Shimokawa researched and developed Computer Go AI in his master course at the University of Electro-Communications, Tokyo, Japan. He has been with SQUARE

ENIX since 2013. At present, he is developing AI in *FINAL FANTASY XV* and was part of the team that developed Luminous Studio Tools.

Youji Shirakami has worked as a programmer in SQUARE ENIX since 2005. Before that, he worked as an engineer for transportation systems. He was engaged in the development of *Seiken Densetsu FRIENDS of MANA* (Mobile), *KINGDOM HEARTS* (Mobile), and *FINAL FANTASY TYPE-0* (PSP). His current tasks include implementation of the AI Graph Editor and character AI development.

Marius Stanescu is a PhD candidate at the University of Alberta, Edmonton, Canada. He completed his MSc in artificial intelligence at University of Edinburgh, Edinburgh, the United Kingdom, in 2011, and became a researcher at the Center of Nanosciences for Renewable & Alternative Energy Sources of University of Bucharest, Bucharest, Romania, in 2012. Since 2013, he has been helping organize the AIIDE StarCraft Competition. Marius' main areas of research include machine learning, AI, and RTS games.

Ben Sunshine-Hill is the lead developer of Havok AI. He holds a PhD in computer science from the University of Pennsylvania, Philadelphia, Pennsylvania, for his work in perceptually driven simulation. He once saw a really cool-looking cow.

Joudan Tatsuhiro has been with SQUARE ENIX since 2011. He was engaged in the development at Luminous Studio, especially for tools. He has been developing buddy AI within the *FINAL FANTASY XV* team, especially for the meta-AI system that controls them.

William van der Sterren is an AI consultant for games and simulations at CGF-AI. He worked on the AI of Guerrilla Games' *Killzone* and *Shellshock Nam'67* games. William has spoken at the Game Developer Conference and AIGameDev conference, and has contributed chapters to both the *Game Programming Gems* and *AI Game Programming Wisdom* series. His interest is in creating tactical behaviors, from tactical path-finding and terrain analysis to squad behaviors and company level maneuver planning. William holds an MSc in computer science from University of Twente and a PDEng Software Technology from Eindhoven University of Technology.

Will Wilson recently founded Indefiant Ltd. in order to focus on developing software and consulting for improving the iteration times and reducing costs in game development and testing. His 10 years in the games industry includes being lead programmer at Firefly Studios and senior programmer at Crytek, where he worked on *Crysis 2* and the *Crysis* console conversion. At Crytek he developed the SoftCoding implementation for the CryENGINE 3 for use in *Ryse* and *Crysis 3*.

Takanori Yokoyama has worked as a game programmer in the game industry since 2004. He has been especially interested in game AI and implemented it for many game titles including *ENCHANT ARMS* (2006, Xbox360), *CHROME HOUNDS* (2006, Xbox360), and *Demon's Souls* (2009, PlayStation3) developed by FROM SOFTWARE. He is now working as an AI engineer at SQUARE ENIX.

Robert Zubek is a game developer and cofounder at SomaSim, a Chicago-based indie studio founded in 2013 to create simulation games. Previously, he built large-scale online social games at Zynga, MMO game and analytics infrastructure at Three Rings Design, and console games at Electronic Arts/Maxis. Before joining the industry, he specialized in artificial intelligence and robotics research. Robert holds a PhD in computer science from Northwestern University, Evanston, Illinois, where he also received his previous computer science degrees.

Introduction

Steve Rabin's ***Game AI Pro 360: Guide to Architecture*** gathers all the cutting-edge information from his previous three Game AI Pro volumes into a convenient single source anthology covering game AI architecture.

This volume of carefully curated content is complete with articles and real-life case studies by leading game AI programmers that further explores modern architecture such as behavior trees and share architecture used in top games such as *Final Fantasy XV*, the *Call of Duty* series and the *Guild War* series.

This book, as well as each volume in the *Game AI Pro* series, is a testament to the generous community of game AI developers as well as the larger game development community. Everyone involved in this effort believes that sharing information is the single best way to rapidly innovate, grow and develop. Right now, the game AI community is larger than ever and we invite you to discover all the wonderful resources that are available.

In addition to reading about new game AI techniques in the *Game AI Pro* series, there are annual conferences, which are academic and developer-centric, all over the globe. Organized by developers, there is the Game AI summit at GDC in San Francisco each year and the game/AI conference in Europe. Organized by academia, there is the AAAI conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE) and the IEEE Conference on Computational Intelligence and Games. Outside of events, there are two communities that have also sprung up to help developers. The game AI Programmers Guild is a free professional group with more than 500 worldwide members (www.gameai.com) and there is a wonderful community of hobbyists and professionals at www.AIgameDev.com. We warmly welcome you to come and hang out with us at any one of these conferences or participate in one of the online communities.

Web Materials

Example programs and source code to accompany some of the chapters are available at http://www.gameaipro.com.

General System Requirements

The following is required to compile and execute the example programs:

- The DirectX August 2009 SDK
- DirectX 9.0 compatible or newer graphics card
- Windows 7 or newer
- Visual C++ .NET 2008 or newer

Updates of the example programs and source code will be updated as needed.

1

Behavior Selection Algorithms

An Overview

Michael Dawe, Steve Gargolinski, Luke Dicken, Troy Humphreys, and Dave Mark

1.1 Introduction
1.2 Finite-State Machines
1.3 Hierarchical Finite-State Machines
1.4 Behavior Trees
1.5 Utility Systems
1.6 Goal-Oriented Action Planners
1.7 Hierarchical Task Networks
1.8 Conclusion

1.1 Introduction

Writing artificial intelligence systems for games has become increasingly complicated as console gamers demand more from their purchases. At the same time, smaller games for mobile platforms have burst onto the scene, making it important for an AI programmer to know how to get the best behavior out of a short frame time.

Even on complicated games running on powerful machines, NPCs can range from simple animals the player might run past or hunt to full-fledged companion characters that need to stand up to hours of player interaction. While each of these example AIs may follow the Sense–Think–Act cycle, the "think" part of that cycle is ill-defined. There are a variety of algorithms to choose from, and each is appropriate for different uses. What might be the best choice to implement a human character on the latest consoles might not be suitable for creating an adversarial player for a web-based board game.

This article will present some of the most popular and proven decision-making algorithms in the industry, providing an overview of these choices and showing when each might be the best selection to use. While it is not a comprehensive resource, hopefully it will prove a good introduction to the variety of algorithmic choices available to the AI programmer.

1.2 Finite-State Machines

Finite-state machines (FSMs) are the most common behavioral modeling algorithm used in game AI programming today. FSMs are conceptually simple and quick to code, resulting in a powerful and flexible AI structure with little overhead. They are intuitive and easy to visualize, which facilitates communication with less-technical team members. Every game AI programmer should be comfortable working with FSMs and be aware of their strengths and weaknesses.

An FSM breaks down an NPC's overall AI into smaller, discrete pieces known as *states*. Each state represents a specific behavior or internal configuration, and only one state is considered "active" at a time. States are connected by *transitions*, directed links responsible for switching to a new active state whenever certain conditions are met.

One compelling feature of FSMs is that they are easy to sketch out and visualize. A rounded box represents each state, and an arrow connecting two boxes signifies a transition between states. The labels on the transition arrows are the conditions necessary for that transition to fire. The solid circle indicates the initial state, the state to be entered when the FSM is first run. As an example, suppose we are designing an FSM for an NPC to guard a castle, as in Figure 1.1.

Our guard NPC starts out in the *Patrol* state, where he follows his route and keeps an eye on his part of the castle. If he hears a noise, then he leaves *Patrol* and moves to *Investigate* the noise for a bit before returning to *Patrol*. If at any point he sees an enemy, he will move into *Attack* to confront the threat. While attacking, if his health drops too low, he'll *Flee* to hopefully live another day. If he defeats the enemy, he'll return to *Patrol*.

While there are many possible FSM implementations, it is helpful to look at an example implementation of the algorithm. First is the `FSMState` class, which each of our concrete states (*Attack*, *Patrol*, etc.) will extend:

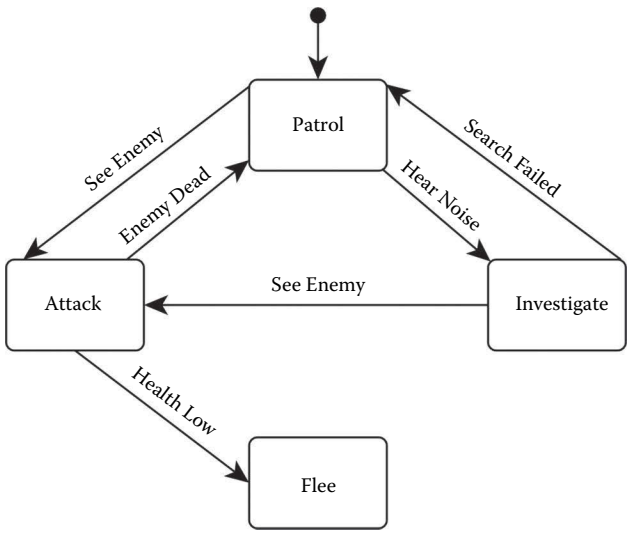

Figure 1.1

This FSM diagram represents the behavior of a guard NPC.

```
class FSMState
{
    virtual void onEnter();
    virtual void onUpdate();
    virtual void onExit();
    list<FSMTransition> transitions;
};
```

Each `FSMState` has the opportunity to execute logic at three different times: when the state is entered, when it is exited, and on each tick when the state is active and no transitions are firing. Each state is also responsible for storing a list of `FSMTransition` objects, which represent all potential transitions out of that state.

```
class FSMTransition
{
    virtual bool isValid();
    virtual FSMState* getNextState();
    virtual void onTransition();
}
```

Each transition in our graph extends from `FSMTransition`. The `isValid()` function evaluates to true when this transition's conditions are met, and `getNextState()` returns which state to transition to when valid. The `onTransition()` function is an opportunity to execute any necessary behavioral logic when a transition fires, similar to `onEnter()` in `FSMState`.

Finally, the `FiniteStateMachine` class:

```
class FiniteStateMachine
{
    void update();
    list<FSMState> states;
    FSMState* initialState;
    FSMState* activeState;
}
```

The `FiniteStateMachine` class contains a list of all states in our FSM, as well as the initial state and the current active state. It also contains the central `update()` function, which is called each tick and is responsible for running our behavioral algorithm as follows:

- Call `isValid()` on each transition in `activeState.transtitions` until `isValid()` returns true or there are no more transitions.
- If a valid transition is found, then:
  - Call `activeState.onExit()`
  - Set `activeState` to `validTransition.getNextState()`
  - Call `activeState.onEnter()`
- If a valid transition is not found, then call `activeState.onUpdate()`

With this structure in place, it's a matter of setting up transitions and filling out the `onEnter()`, `onUpdate()`, `onExit()`, and `onTransition()` functions to produce the desired AI behavior. These specific implementations are entirely design dependent. For example, say our Attack state triggers some dialogue, "There he is, get him!" in

`onEnter()` and uses `onUpdate()` to periodically choose tactical positions, move to cover, fire on the enemy, and so on. The transition between *Attack* and *Patrol* can trigger some additional dialogue: "Threat eliminated!" in `onTransition()`.

Before starting to code your FSM, it can be helpful to sketch a few diagrams like the one in Figure 1.1 to help define the logic of the behaviors and how they interconnect. Start writing the code once the different states and transitions are understood. FSMs are flexible and powerful, but they only work as well as the thought that goes into developing the underlying logic.

1.3 Hierarchical Finite-State Machines

FSMs are a useful tool, but they do have weaknesses. Adding the second, third, or fourth state to an NPC's FSM is usually structurally trivial, as all that's needed is to hook up transitions to the few existing required states. However, if you're nearing the end of development and your FSM is already complicated with 10, 20, or 30 existing states, then fitting your new state into the existing structure can be extremely difficult and error-prone.

There are also some common patterns that FSMs are not well-equipped to handle, such as situational behavior reuse. To show an example of this, Figure 1.2 shows a night watchman NPC responsible for guarding a safe in a building.

This NPC will simply patrol between the front door and the safe forever. Suppose a new state called *Conversation* is to be added that allows our night watchman to respond to a cell phone call, pause to have a brief conversation, and return to his patrol. If the watchman is in *Patrol to Door* when the call comes in, then we want him to resume patrolling to the door when the conversation is complete. Likewise, if he is in *Patrol to Safe* when the phone rings, he should return to *Patrol to Safe* when transitioning out of *Conversation*.

Since we need to know which state to transition back to after the call, we're forced to create a new *Conversation* state each time we want to reuse the behavior, as shown in Figure 1.3.

In this simple example we require two *Conversation* behaviors to achieve the desired result, and in a more complicated FSM we might require many more. Adding additional states in this manner every time we want to reuse a behavior is not ideal or elegant. It leads to an explosion of states and graph complexity, making the existing FSM harder to understand and new states ever more difficult and error-prone to add.

Thankfully, there is a technique that will alleviate some of these structural issues: the Hierarchical Finite-State Machine (HFSM). In an HFSM, each individual state can be an entire state machine itself. This technique effectively separates one state machine into multiple state machines arranged in a hierarchy.

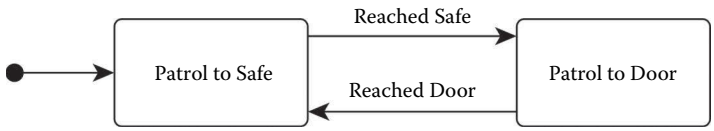

Figure 1.2

This FSM diagram represents the behavior of a night watchman NPC.

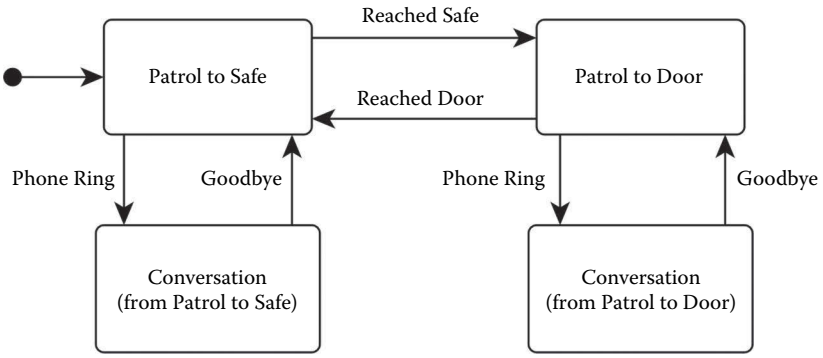

Figure 1.3

Our night watchman FSM requires multiple instances of the Conversation state.

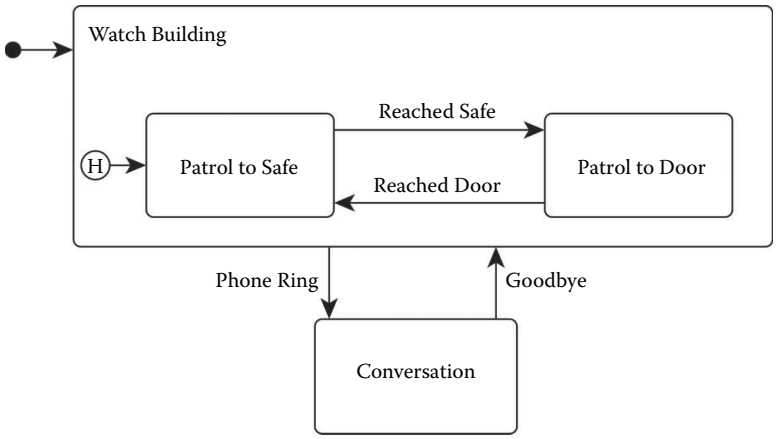

Figure 1.4

An HFSM solves the problem of duplicate Conversation states.

Returning to the night watchman example, if we nest our two *Patrol* states into a state machine called *Watch Building*, then we can get by with just one *Conversation* state, as shown in Figure 1.4.

The reason this works is that the HFSM structure adds additional hysteresis that isn't present in an FSM. With a standard FSM, we can always assume that the state machine starts off in its initial state, but this is not the case with a nested state machine in an HFSM. Note the circled "H" in Figure 1.4, which points to the "history state." The first time we enter the nested *Watch Building* state machine, the history state indicates the initial state, but from then on it indicates the most recent active state of that state machine.

Our example HFSM starts out in *Watch Building* (indicated by the solid circle and arrow as before), which chooses *Patrol to Safe* as the initial state. If our NPC reaches the safe and transitions into *Patrol to Door*, then the history state switches to *Patrol to Door*. If the NPC's phone rings at this point, then our HFSM exits *Patrol to Door* and *Watch*

Building, transitioning to the *Conversation* state. After *Conversation* ends, the HFSM will transition back to *Watch Building* which resumes in *Patrol to Door* (the history state), not *Patrol to Safe* (the initial state).

As you can see, this setup achieves our design goal without requiring duplication of any states. Generally, HFSMs provide much more structural control over the layout of states, allowing larger, complex behaviors to be broken down into smaller, simpler pieces.

The algorithm for updating an HFSM is similar to updating an FSM, with added recursive complexity due to the nested state machines. Pseudocode implementation is fairly complicated, and beyond the scope of this overview article. For a solid detailed implementation, check out Section 5.3.9 in the book *Artificial Intelligence for Games* by Ian Millington and John Funge [Millington and Funge 09].

FSMs and HFSMs are incredibly useful algorithms for solving a wide variety of problems that game AI programmers typically face. As discussed there are many pros to using an FSM, but there are also some cons. One of the major potential downsides of FSMs is that your desired behavior might not fit into the structure elegantly. HFSMs can help alleviate this pressure in some cases, but not all. For example, if an FSM suffers from "transition overload" and hooks up every state to every other state, and if an HFSM isn't helping, other algorithms may be a better choice. Review the techniques in this article, think about your problem, and choose the best tool for the job.

1.4 Behavior Trees

A *behavior tree* describes a data structure starting from some root node and made up of *behaviors*, which are individual actions an NPC can perform. Each behavior can in turn have child behaviors, which gives the algorithm its tree-like qualities.

Every behavior defines a *precondition*, which specifies the conditions where the agent will execute this behavior, and an *action*, specifying the actual things the agent should do when performing the behavior. The algorithm starts at the root of the tree and examines the preconditions of the behaviors, deciding on each behavior in turn. At each level of the tree, only one behavior can be selected, so if a behavior executes, none of its siblings will be checked, though its children will still be examined. Conversely, if a behavior's precondition does not return true, the algorithm skips checking any of that behavior's children and instead moves onto the next sibling. Once the end of the tree is reached, the algorithm has decided on the highest-priority behaviors to run, and the actions of each are executed in turn.

The algorithm to execute a behavior tree is as follows:

- Make root node the current node
- While current node exists,
  - Run current node's precondition
  - If precondition returns true,
    - Add node to execute list
    - Make node's child current node
  - Else,
    - Make node's sibling current node
- Run all behaviors on the execute list

The real strength of a behavior tree comes from its simplicity. The base algorithm can be implemented quickly due to its straightforward nature. Since trees are stateless, the algorithm doesn't need to remember what behaviors were previously running in order to determine what behaviors should execute on a given frame. Further, behaviors can (and should) be written to be completely unaware of each other, so adding or removing behaviors from a character's behavior tree do not affect the running of the rest of the tree. This alleviates the problem common with FSMs, where every state must know the transition criteria for every other state.

Extensibility is also an advantage with behavior trees. It is easy to start from the base algorithm as described and start adding extra functionality. Common additions are behavior `on_start/on_finish` functions that are run the first time a behavior begins and when it completes. Different behavior selectors can be implemented as well. For example, a parent behavior could specify that instead of choosing one of its children to run, each of its children should be run once in turn, or that one of its children should be chosen randomly to run. Indeed, a child behavior could be run based on a utility system-type selector (see below) if desired. Preconditions can be written to fire in response to events as well, giving the tree flexibility to respond to agent stimuli. Another popular extension is to specify individual behaviors as nonexclusive, meaning that if their precondition is run, the behavior tree should keep checking siblings at that level.

A behavior tree, though simple and powerful, is not always the best choice for a selection algorithm. Since the tree must run from the root every time behaviors are selected, the running time is generally greater than that of a finite-state machine. Additionally, the naïve implementation can have a large number of conditional statements, which can be very slow, depending on your target platform. On the other hand, evaluating every possible behavior in the tree may be slow on others where processing power is the limiting factor. Either approach can be a valid implementation of the algorithm; so the programmer would have to decide what is best.

Since behaviors themselves are stateless, care must be taken when creating behaviors that appear to apply memory. For example, imagine a citizen running away from a battle. Once well away from the area, the "run away" behavior may stop executing, and the highest-priority behavior that takes over could take the citizen back into the combat area, making the citizen continually loop between two behaviors. While steps can be taken to prevent this sort of problem, traditional planners can tend to deal with the situation more easily.

1.5 Utility Systems

Much of AI logic—and, for that matter, computer logic—is based on simple Boolean questions. For example, an agent may ask "Can I see the enemy?" or "Am I out of ammunition?" These are purely "yes or no" questions. The decisions that come out of Boolean questions are often just as polarized. As we saw in the prior architectures, the results of these questions are often mapped directly to a single action. For instance,

```
if (CanSeeEnemy())
{
    AttackEnemy();
}
```

```
if (OutOfAmmo())
{
    Reload();
}
```

Even when multiple criteria are combined, Boolean equations tend to lead to a very discrete result set.

```
if (OutOfAmmo() && CanSeeEnemy())
{
    Hide();
}
```

Many aspects of decision making aren't quite as tidy, however. There are numerous questions that can be asked where a "yes or no" answer is not appropriate. For example, we may want to consider how far away the enemy is, how many bullets I have left, how hungry I am, how wounded I am, or any number of continuous values. Correspondingly, these continuous values can be mapped over into *how much* I want to take an action rather than simply whether to take the action or not. A utility-based system measures, weighs, combines, rates, ranks, and sorts out many considerations in order to decide the *preferability* of potential actions. Using the above example as a guide, we could assess how *strongly* we want (or need!) to attack, reload, hide, etc.

While utility techniques can be used to supplement the transition logic of other architectures, it is very possible to build an entire decision engine based on utility. In fact, there are times when building a utility-based AI is far preferable to other methods. These might include games where there are many possible actions, and either there isn't a single "right" answer or the selection of a preferable action might be based on a large number of competing inputs. In these cases, we are going beyond simply using utility to measure or rate something. Instead, we are using it to drive the actual decision mechanism as well. Another way of stating it is that, rather than saying "This is the one action you will do," the utility-based system *suggests*, "Here are some possible options that you might want to do."

One well-documented example of this is the use of utility in *The Sims*. In these games, the agents (i.e., the actual "Sims") take information from their environment and combine it with their own internal state to arrive at a preferability score for each potential action. For example, the fact that I am "very hungry" combined with the availability of "poor food" would certainly be more attractive than if I was only "a little hungry." Additionally, the proximity of "spectacular" food might still make for a high priority even if I was only "a little hungry." Note that the descriptors "spectacular," "rather," "poor," and "a little" would actually be numbers between some set minimum and a maximum. (A typical method to use is a floating point number between 0 and 1.)

When it is time to select a new action (either because the current one is finished or through some sort of interrupt system), some method is used to select from among the candidates. For example, the scores for the potential actions could be sorted so that we can simply select the "most appropriate" action—that is, the one with the highest score. An alternate way is to use the scores to seed a weighted random selection. By casting a random number against these weighted probabilities, the most preferable actions have a higher chance of being selected. As an action's suitability goes up, its score goes up, as does its chance of being selected.

Another example of where utility-based architectures might be preferable to other architectures is RPGs. Often in these games, the options that an agent has are varied and possibly only subtly better or worse, given the situation. For instance, selecting what weapon, spell, item, or action should be taken given the type of enemy, the agent's status, the status of the player, etc., can be a complicated balancing act.

Another wheelhouse of utility architectures is any game system with an economic decision layer. The question of units or buildings to construct in a real-time strategy game, for example, is a juggling act of costs, times, and often many axes of priority (e.g., "offense" or "defense"). An architecture based on utility can often be more adaptable to changing game situations. As such, it can recover better from being disrupted than can more scripted models, which can suffer from either being hopelessly confused or can simply trundle along as if nothing ever happened.

The primary reason for this adaptability is that preferability scores are highly dynamic. As the game situation changes—either through a change in the environment or a change in state of the agent—the scores for most (if not all) of the actions will change. As the action scores change, so does their likelihood of being selected as a "reasonable" action. The resulting ebb and flow of action scores—especially when combined with a weighted random selection—often leads to very dynamic emergent behavior.

On the other hand, unlike the architectures that use Boolean-transitioned decision logic, utility systems are often somewhat unpredictable. Because the selections are based on how much the actions "make sense" in a given situation and context, however, the actions should tend to look *reasonable*. This unpredictability has benefits and drawbacks. It can improve believability because the variety of actions that could occur in a given situation can make for far more natural-looking agents rather than the predictably robotic if/then-based models. While this is desirable in many situations, if your design calls for specific behaviors at very certain moments, you must make a point to override the utility calculations with more scripted actions.

Another caveat to using utility-based architecture is that all the subtlety and responsiveness that you gain often comes at a price. While the core architecture is often relatively simple to set up, and new behaviors can be added simply, they can be somewhat challenging to tune. Rarely does a behavior sit in isolation in a utility-based system. Instead, it is added to the pile of all the other potential behaviors with the idea that the associated mathematical models will encourage the appropriate behaviors to "bubble to the top." The trick is to juggle all the models to encourage the most reasonable behaviors to shine when it is most appropriate. This is often more art than science. As with art, however, the results that are produced are often far more engaging than those generated by using simple science alone.

For more on utility-based systems, see the article in this book, *An Introduction to Utility Theory* [Graham 13] and the book *Behavioral Mathematics for Game AI* [Mark 09].

1.6 Goal-Oriented Action Planners

Goal-Oriented Action Planning (GOAP) is a technique pioneered by Monolith's Jeff Orkin for the game *F.E.A.R.* in 2005, and has been used in a number of games since, most recently for titles such as *Just Cause 2* and *Deus Ex: Human Revolution*. GOAP is derived from the Stanford Research Institute Problem Solver (STRIPS) approach to AI which was first developed in the early 1970s. In general terms, STRIPS (and GOAP) allows an AI system

to create its own approaches to solving problems by being provided with a description of how the game world works—that is, a list of the actions that are possible, the requirements before each action can be used (called "preconditions"), and the effects of the action. The system then takes a symbolic representation of the initial state of the world and some set of objective facts that need to be achieved. In GOAP these objectives are typically chosen from a predetermined set of goals that an NPC may want to achieve, chosen by some method such as priority or state transition. The planning system can then determine a sequence of actions that will allow the agent that it is controlling to change the world from the original state into a state that contains the facts that need to be true to satisfy its current goals. In classical planning this would ideally be the critical path to the target state, and that target would be the most easily reachable state that contained all of the objective facts.

GOAP works by "*backwards chaining search,*" which is a fancy phrase which means starting with the goals you want to achieve, working out what actions are required for those to happen, then working out what needs to happen in order to achieve the preconditions of the actions you just identified and so on. You continue to work backwards in this fashion until you arrive at the state you started from. It's a fairly traditional approach, which has fallen out of favor in the scientific world, replaced by "*forwards chaining search*" which relies on heuristic search, pruning, and other tricks. Backwards search is a solid workhorse, however, and although it's less elegant, it's far easier to understand and implement than more modern techniques.

Backwards chaining search works in the following manner:

- Add the goal to the outstanding facts list
- For each outstanding fact
  - Remove this outstanding fact
  - Find the actions that have the fact as an effect
  - If the precondition of the action is satisfied,
    - Add the action to the plan,
    - Work backwards to add the now-supported action chain to the plan
  - Otherwise,
    - Add the preconditions of the action as outstanding facts

One final interesting aspect of GOAP is that it allows "context preconditions" that are ignored by the planning system, but must be satisfied at run-time in order for an action to be executed. This allows for reasoning to bypass certain aspects of the world that cannot be easily represented symbolically—such as ensuring line of sight to a target before beginning to fire—while ensuring that by accessing information not made available during planning (to ensure the search remains tractable), these constraints can be met. This allows the plan GOAP generates to be somewhat flexible, and the actions it calls for apply more at a tactical level than at the most basic level of execution. That is, the plan tells you *what* to do, but not necessarily *how* to do it. For example, detailed instructions such as how to establish a line of sight to begin shooting are omitted and can be handled more reactively.

Let's suppose we have a typical NPC soldier character, whose goal is to kill another character. We can represent this goal as `Target.Dead`. In order for the target to die, the character needs to shoot him (in a basic system). A precondition of shooting is having a weapon equipped. Assuming our character doesn't have one, we now need an action that

can give the character a weapon, perhaps by drawing one from a holster. This, of course, has its own precondition—that there is a weapon available in the character's inventory. If this is the case, we have just created a simple plan of drawing the weapon and then shooting. What if the character doesn't have a weapon? Then our search would have to find a way to get one. If that isn't possible the search can backtrack and look for alternatives to the shoot action. Perhaps there is a mounted weapon nearby that could be used to provide the `Target.Dead` effect, or even a vehicle that we can use for running over the target. In either case, it's clear that by providing a comprehensive set of action choices of what *can* be done in the world, we can leave it up to the character to decide what *should* be done, letting dynamic and interesting behaviors emerge naturally, rather than having to envisage and create them during development.

Finally, consider a game in which weapons have a maximum range. As a context precondition, we can say that the target must be within that range. The planner won't spend time in its search trying to make this true—it can't, as it would involve reasoning about how the target might move and so on—but it either won't fire its weapon until the condition is true, or it will instead use an alternative tactic such as a different weapon with a longer range.

There's a lot to like about an approach to NPC control based on automated planning. It streamlines the development process by allowing designers to focus on creating simple components that will self-assemble into behaviors, and it also allows for "novel" solutions, which may never have been anticipated by the team, often making for excellent anecdotes that players will re-tell. GOAP itself remains the lowest hanging fruit of what automated planning can provide and, from a purely scientific point of view, the state of the art has progressed significantly since it was developed. With that said, it can still be a very powerful technique when used correctly, and provides a good, adaptable starting point for specific customization.

It is worth noting that these kinds of approaches that adopt a character-centric view of intelligence remove a lot of the authorial and directorial control from the development team. Characters that can "think" for themselves can become loose cannons within the game world, creating plans that, while valid for achieving the character's goals, do not achieve the broader goals of creating immersive and engaging experiences, and this can then potentially disrupt cinematic set-pieces if, for example, a soldier's plan doesn't take him past the conveniently placed Big Red Barrel.

While it's possible to avoid these kinds of issues by using knowledge engineering techniques and representational tricks, it isn't as straightforward as with architectures such as behavior trees, which would allow the desired behavior to be injected directly into the character's decision logic. At the same time, a GOAP approach is significantly easier to design than one based around hierarchical task networks, since in GOAP you just need to describe the mechanics of the objects within a world.

GOAP and similar techniques are not silver bullet solutions, but in the right circumstances they can prove to be very powerful in creating realistic behaviors and immersive-feeling characters that players can fully engage with.

1.7 Hierarchical Task Networks

Though GOAP is perhaps the best-known game planner, other types of planners have gained popularity as well. One such system, *hierarchical task networks* (HTN), has been used in titles such as Guerrilla Games' *KillZone 2* and High Moon Studios' *Transformers:*

Fall of Cybertron. Like other planners, HTN aims to find a plan for the NPC to execute. Where it differs is how it goes about finding that plan.

HTN works by starting with the initial world state and a root task representing the problem we are looking to solve. This high-level task is then decomposed into smaller and smaller tasks until we end up with a plan of tasks we can execute to solve our problem. Each high-level task can have multiple ways of being accomplished, so the current world state will be used to decide which set of smaller tasks the high-level task should be decomposed into. This allows for decision making at multiple levels of abstraction.

As opposed to *backward* planners like GOAP, which start with a desired world state and move backwards until it reaches the current state world state, HTN is a *forward* planner, meaning that it will start with the current world state and work towards a desired solution. The planner works with several types of primitives, starting with the *world state*. The world state represents the state of the problem space. An example in game terms might be an NPC's view of the world and him in it. This world state is broken up into multiple properties such as his health, his stamina, enemy's health, enemy's range, and the like. This knowledge representation would allow the planner to reason about what to do.

Next, we have two different types of tasks: *primitive tasks* and *compound tasks*. A primitive task is an actionable thing that can be done to solve a problem. In game terms, this could be *FireWeapon*, *Reload*, and *MoveToCover*. These tasks are able to affect the world state, such as how the *FireWeapon* task would use ammo and the *Reload* task would refill the weapon. *Compound task*s are higher level tasks that can be accomplished in different ways, described as *methods*. A method is a set of tasks that can accomplish the compound task, along with preconditions determining when a method may be used. Compound tasks allow HTN to reason about the world and decide which course of action to take.

Using compound tasks, we can now build an HTN domain. The domain is a large hierarchy of tasks that represent all the ways of solving our problem, such as how to behave as an NPC of some type. The following pseudocode shows how a plan is built.

- Add the root compound task to our decomposing list
- For each task in our decomposing list
  - Remove task
  - If task is compound
    - Find method in compound task that is satisfied by the current world state
    - If a method is found, add method's tasks to the decomposing list
    - If not, restore planner to the state before the last decomposed task
  - If task is primitive
    - Apply task's effects to the current world state
    - Add task to the final plan list

As mentioned, HTN planners start with a very high-level root task and continuously decompose it into smaller and smaller tasks. This decomposition is steered with each compound task's set of methods by comparing each method's conditions with the current world state. When we finally come across a primitive task, we add it to our final plan. Since each primitive task is an actionable step, we can apply its effects to the world state, essentially moving forward in time. Once the decomposing list is empty we will either have a valid plan or have backed out entirely leaving us with no plan.

To demonstrate how an HTN works, suppose a game has a Soldier NPC that needs its AI written. The root compound task might be named *BeSoldierTask*. Next, the soldier should behave differently if he had an enemy to attack or not. Therefore, two methods are needed to describe what to do in these cases. In the case where an enemy is present, the *BeSoldierTask* would decompose using the method that required that condition. The method's task in this case would be *AttackEnemyTask*. This task's methods define the different ways that the soldier could attack. For example, the soldier could shoot from a cover position if he has ammunition for his rifle. If he didn't have ammo for his firearm, he could charge the enemy and attack him with his combat knife. Writing these give *AttackEnemyTask* two methods to complete the task.

The more we drill down on a soldier's behavior, the more the hierarchy forms and refines. The structure of the domain fits naturally in how one might describe the behavior to another person.

Since HTNs describe behavior using a hierarchical structure, building and reasoning about characters is done in a natural way, allowing designers to more easily read through HTN domains, assisting with collaboration between programming and design. Like other planners, the work that is actually done by the AI is kept in nice modular primitive tasks, allowing a lot of reuse across different AI characters.

Since HTN is a search through a graph, the size of your graph will affect search times, but there are two ways to control search size. First, the method's conditions can be used to cull entire branches of the hierarchy. This happens naturally as behaviors are built. Second, having partial plans can defer complex calculations until plan execution. For example, consider the compound task *AttackEnemy.* One method might have the subtasks *NavigateToEnemy* followed by *MeleeEnemy*. *NavigateToEnemy* requires pathing calculations, which can not only be costly, but could be affected by the state of the world, which might change between planning and execution. To utilize partial plans, split these two tasks into two methods, rather than one method with subtasks: *NavigateToEnemy*, if the enemy is out of range, and *MeleeEnemy* when in range. This allows us to only form a partial plan of *NavigateToEnemy* when the enemy is out of range, shortening our search time.

One other note is that the user needs to build the network for HTN to work. This is a double-edged sword when comparing it to GOAP-style planners. While this allows the designer to be very expressive in the behavior they are trying to achieve, it removes the NPC's ability to build plans that the designer might not have thought of. Depending on the game you are building, this can be considered either a strength or a weakness.

1.8 Conclusion

With such a wide variety of behavior selection algorithms available, it is imperative that the AI programmer have knowledge of each tool in the toolbox in order to best apply each to a given situation. Which algorithm is best for a given NPC can depend on the game, the knowledge state of the NPC, the target platform, or more. While this is not a comprehensive treatment on every available option, knowing a bit about where to start with the options can be invaluable. By taking time to think carefully about the needs of the game, an AI system can be crafted to give the best player experience while maintaining the balance between development time and ease of creation.

References

[Graham 13] R. Graham. "An Introduction to Utility Systems." In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.

[Mark 09] D. Mark. *Behavioral Mathematics for Game AI*. Boston, MA: Cengage Learning, 2009.

[Millington and Funge 09] I. Millington and J. Funge. *Artificial Intelligence for Games*. Burlington, MA: Morgan Kaufmann, 2009, pp. 318–331.

2

Structural Architecture—Common Tricks of the Trade

Kevin Dill

2.1 Introduction
2.2 Definitions
2.3 Common Architectures
2.4 Hierarchical Reasoning
2.5 Option Stacks
2.6 Knowledge Management
2.7 Modularity
2.8 Conclusion

2.1 Introduction

When discussing game AI, developers often get hyper-focused on advocating for a particular approach or a particular architecture, but many of the problems that make AI programming so hard appear regardless of the architecture you use, and many of the most common solutions have been reinvented time after time in architecture after architecture. For example, we can use *hierarchy* to divide up the AI logic, simplifying both configuration and execution. We can use *option stacks* to allow one option to temporarily suspend another, without changing the overall plan of attack. We can use a *blackboard* to share information and ideas between AI components or between characters. We can *move the intelligence* into the objects, the terrain, the abilities, or the events in order to more logically divide the code and data, and to improve our ability to extend the game. (Some games, such as *The Sims* franchise, have used this approach to ship downloadable content or even entire expansion packs without a change to the executable.) Finally, we can use modularity to extract reusable pieces of AI logic, thus simultaneously eliminating duplicate code and enabling the AI's author to think at a coarser level of granularity.

In the remainder of this chapter we describe each of these ideas. The intent is to provide enough information to paint the big picture, to spark new ideas and solutions to problems that you, the reader, might face, but not to dig deeply into the technical details of any given solution. For that, we refer you to the bibliography and the references contained therein.

2.2 Definitions

The ideas that we will address in this paper are fairly universal, but there aren't necessarily universal terms to describe them—and in some cases, terms mean one thing to one person and something different to another. As a result, this section will define terms that might not be clear, so that we can have a common vocabulary to work with as we forge ahead.

The *AI architecture* is the underlying code—typically in C++—that controls the process by which the AI evaluates the situation and makes a decision. Several examples of architectures will be given below, and the other articles in this section describe many of them in detail.

A *configuration* is the behavioral specification that maps from specific sensory data (i.e., specific inputs) to specific decisions or actions (i.e., specific outputs). For example, the configuration for a First-Person Shooter (FPS) character would contain all of the logic necessary to decide what that character should do at each point in the game—e.g., should it attack, should it run away, should it reload its weapon, and so on. The configuration is built on top of the architecture and usually specified in data (i.e., XML or similar).

We will often refer to the thing that an AI controls as the *character*. Not all AIs control single characters—for example, you might be working on the AI for an opposing player in a strategy game (which controls many characters), or the AI for a missile in a flying game, or even an AI that controls some aspect of the user interface. You will also hear AI-controlled entities referred to as *agents*, *enemies*, or simply as *AIs*. The generic term *character*, however, is a convenient term, and most AI does apply to characters, so we will stick with it in this chapter.

It is often the case that a character will contain more than one decision-maker. For example, a character might have one algorithm that selects which weapon to use, another that selects the target to shoot at, and a third that decides what emotion he should display. We refer to these decision makers as *reasoners*. Reasoners may have clear cut responsibilities (such as those described above), or they may be organized in a more ad hoc fashion (such as the selectors in a behavior tree [Isla 05]).

Similarly, we will refer to the things that reasoners choose as *options*. A reasoner functions by first evaluating the situation, and then selecting one or more options to execute. When a reasoner picks a new option and starts executing it, we say that it has *selected* the option, and when it stops executing an option we say that the option is *deselected*. These options might be *actions* (that is, physical things that the AI does), but they might also be more abstract. For example, a character's emotional reasoner might simply change the AI's internal state (perhaps by setting an enumerated value to the appropriate state—`eHappy`, `eSad`, `eAngry`, etc.—or by assigning an intensity value to each emotion) so that other reasoners can then select appropriate actions (smiling, frowning, attacking the player, etc.) based on that state.

2.3 Common Architectures

Although the techniques described here are intended to be architecture-agnostic, it is often useful to discuss a specific architecture when describing them. All of these architectures are described in detail elsewhere, so we give just a very high level definition of each one.

Scripting is perhaps the most basic architecture possible. In it, the designer or AI programmer specifies the sequence of options that the AI will select and when they will be selected (e.g., wait 48 seconds, and then spawn three units and attack the player). Scripts may include very simple sensory input, such as trigger zones, but in general the idea is that every decision is fully specified by the AI's author [Berger et al. 02].

Finite-state machines (FSMs) were once the most popular game AI architecture, but have been largely replaced by behavior trees in recent years. An FSM is a collection of states and transitions. The states represent the options that the reasoner can select, whereas the transitions represent the conditions under which the AI will change from one state to another. For example, a very simple first-person shooter (FPS) character might have four states: *Attack*, *Flee*, *Reload*, and *Search for Enemy*. The *Attack* state would contain transitions to *Flee* (which fires if the character is nearly dead), *Reload* (which fires when the character is out of ammo), and *Search for Enemy* (which fires when the character loses sight of the enemy) [Buckland 05, Rabin 00].

A *rule-based AI* is one that consists of a sequence of predicate-option pairs. The AI evaluates the predicate for each rule in order. When it gets to a rule whose predicate is true, it executes the option for that rule and stops evaluating additional rules. Thus, a rule-based AI for our simple FPS character would have four rules. The first would make it flee if its health is low. The second would make it reload if it is out of ammo. The third would make it attack the player if the player is in sight. Finally, the fourth would search for the player [Millington et al. 09a, Nilson 94].

A *utility-based AI* uses a *heuristic function* to assign a floating-point value (typically called a *weight*, *priority*, or *utility*) to each option. It then selects the option to execute based on those values—for example, by taking the option with the highest utility, or by assigning a weight to each option and using that to guide the probability of random selection. Our simple FPS character would still have the same four possible options, but now it would decide which one to do by evaluating the heuristic function for each option and using the resulting values to guide its final selection [Mark 09, Dill 06, Dill et al. 12a, Dill 12c].

Planners such as *goal-oriented action planners (GOAP)* or *hierarchical task-network (HTN) planners* build a sequence of options that will get them to some goal state. For example, our FPS character might have the goal *Enemy Dead*. It would search through its possible options and the ways in which they can change the state of the world in order to find a sequence of options which will get it into the goal state. That plan might be something like *Search for Enemy–Attack–Reload–Attack* (if it expects two magazines of ammo to be enough to get the job done). Planners typically have the ability to replan if the situation changes. So if our character finds itself nearly dead then it might replan with a new goal, such as *Don't Die*. This new plan might have only a single option: *Flee* [Orkin 04, Gorniak et al. 07].

The *behavior tree (BT)* architecture is a bit of a special case, because it is an architecture that can contain other architectures. A BT is a tree of *selectors*, each of which makes a single piece of the overall decision. In their original formulation [Isla 05], the selectors were all exceedingly simple and not really architectures in their own right. However, more recent work has discussed the ability to use nearly any architecture in a selector [Dill 11a], making the behavior tree more of a framework (or meta-architecture) than an architecture in its own right.

2.4 Hierarchical Reasoning

The difficulty of building an AI configuration generally scales worse than linearly with the size of the configuration. In other words, the more situations your AI can handle, the more things it takes into account, the more options it contains, etc., the more complex it is to add yet another of one of those things. The reason for this should be fairly intuitive. Whenever you add something new to your AI you need to put at least some effort into considering how that new thing interacts with each thing already in existence. Thus, the cost of adding a new thing increases the more things you have.

The severity of that increase depends in large part on the architecture you are using. FSMs, for example, scale exponentially because the number of transitions that is exponential on the number of states. This becomes unmanageable very quickly. This is one of the principal reasons that FSMs have largely passed out of use for complex problems. Utility-based AI, on the other hand, only requires you to balance the heuristic functions appropriately, while rule-based AI typically just requires you to place your new rule at the proper place in the list. With that said, even a rule-based AI will become brittle when it contains hundreds or thousands of rules (which is not an unreasonable size for many games).

One common way to address this challenge—an approach that has been applied to nearly every architecture, both in games and in academia—is to break the decision making up hierarchically. That is, have a high-level reasoner that makes the big, overarching decisions, and then one or more lower-level reasoners that handle implementation of the higher-level reasoners' decisions. For example, a high-level reasoner might decide whether to execute a daily schedule of ambient tasks (e.g., get up, get breakfast, go to work, etc.), start a conversation with the player, go into combat, and so forth. Each of those options would contain another reasoner, which decides how to go about accomplishing the goal.

The advantage here is that the complexity of AI configuration scales worse than linearly on the number of options *in a particular reasoner*. To give a sense of the relevance, imagine that the cost of configuring the AI is $O(n^2)$ on the number of options (as it is for FSMs). If we have 25 options, then the cost of configuring the AI is on the order of $25^2 = 625$. On the other hand, if we have five reasoners, each with five options, then the cost of configuring the AI is only $5 \times (5^2) = 125$. Conceptually, this makes sense. When we add a new option, we only need to consider how it relates to other options within the same reasoner—which is much simpler than comparing it to every other option anywhere in the AI.

Examples of this approach abound, from hierarchical FSMs [Millington et al. 09b] to HTN planners [Gorniak et al. 07] and GOAP implementations [Cerpa et al. 08] to strategy game AIs that break the opposing player AI into command hierarchies [Pittman 08]. Behavior trees are perhaps the archetypical example—a BT is really nothing more than a hierarchical infrastructure in which you can place whatever sorts of reasoning architectures best encapsulate the decisions to be made.

2.5 Option Stacks

Most reactive AIs function by evaluating the situation very frequently (often every frame), and deciding what is the best thing to do right at that particular moment. They may have a history of past decisions that guides their choices, they may even have some greater plan

that they're following, but decisions are made moment-to-moment. This is what allows the AI to respond if the situation changes.

Of course, we do want options to be persistent. That is, we don't want the AI to be constantly flip-flopping between different decisions—attacking on one frame, fleeing on the next, and then attacking again on the frame after. Or, for another example, attacking with a shotgun, and then a flamethrower, and then right back to the shotgun after only a frame or two, switching weapons too fast to even get a shot off. That sort of indecisiveness makes the AI look stupid, even if there is a good reason *at that moment* for the decision being made. As we discussed in an earlier chapter [Dill 13], looking stupid is the single worst thing that an AI can do. It breaks the player's suspension of disbelief—that is, their immersion in the experience. As a result, most architectures have some form of *inertia* built in, which keeps the AI doing the same thing unless there is a good reason for the change.

When the AI does change options, one of two possible situations pertains. In most cases, the AI's decision is lasting—that is, it has decided to stop the old option and start a new one, and it's not expecting to go back to what it was doing before. For example, if the AI is in a fight and it kills one enemy, now it can pick a new enemy to attack (or pick something else to do if the fight is over). The decision can be lasting even if the AI wasn't done with the previous option. For example, when an AI decides to flee, that's a lasting decision, even though it typically happens *before* the AI finishes its attack. Regardless, the AI has made a deliberate decision to stop doing what it was doing and do something else instead. In this case, we should stop applying inertia to the deselected option, and in fact may even want to apply a *cooldown* which will prevent us from returning to it for a short period of time.

There are situations, however, when the AI needs to react to an immediate need or opportunity, but once that reaction is complete it should return to its previous option. For example, if an AI needs to reload, it should return to the same action (presumably firing a particular weapon) when it finishes reloading. It wouldn't make sense to reload your shotgun, only to then immediately switch weapons to a flamethrower (or decide to flee). That's not to say that the AI can't change its mind, but it should have to overcome the option's inertia to do so, just as if that option were still executing. Thus, we might switch to the flamethrower immediately after reloading the shotgun—but only if we suddenly spotted some new enemy who is highly vulnerable to fire.

One common trick which has been applied to a great many architectures is to have a stack of currently executing options. This stack is sometimes referred to as a *state stack* [Tozour 04], or a *goal stack* [Cerpa 08], or *subsumption* [Heckel et al. 09], depending on the underlying architecture, but we will simply call it an *option stack*, since that is an architecture-agnostic term. Option stacks allow us to push a new, high priority option on top of the stack, *suspending* the currently executing option but retaining its internal state. When the high priority option completes execution, it will pop itself back off of the stack, and the previously executing option will *resume* as if nothing had ever happened.

There are a myriad of uses for option stacks, and they can often be several levels deep. For example, a high-level strategic reasoner might have decided to send a unit to attack a distant enemy outpost. Along the way, that unit could be ambushed—in which case, it might push a *React to Ambush* option on top of its option stack. While responding to the ambush, one of the characters in the unit might notice that a live grenade has just been thrown at its feet. That character might then push an *Avoid Grenade* option on top of

React to Ambush. Once the grenade has gone off (assuming the character lives) it can pop *Avoid Grenade* off the stack, and *React to Ambush* will resume. Once the enemy ambush is over, it will be popped as well, and the original *Attack* option will resume.

One handy trick is to use option stacks to handle your hit reaction. If a character is hit by an enemy attack (e.g., a bullet), we typically want them to play a visible reaction. We also want the character to stop whatever it was doing while it reacts. For instance, if an enemy is firing their weapon when we hit them, they should not fire any shots while the hit reaction plays. It just looks wrong if they do. Thus, we push an *Is Hit* option onto the option stack, which suspends all previously running options while the reaction plays, and then pop it back off when the reaction is done.

We mentioned this above, but it's worth reemphasizing that the option stack is not meant to *prevent* the AI from changing what it's doing, but simply to preserve its previous state so that can be figured into the decision appropriately. To extend the previous example, imagine that the character was hit in the arm and as a result lost the use of that arm—and therefore could no longer fire its weapon. In that case, it should certainly pick a different option. The option stack simply ensures that the AI has the context of the previous actions available to it so that it can make an informed decision.

Option stacks are surprisingly simple to implement for most architectures. In the AI configuration, each option can specify whether it should suspend the previous option (i.e., push the new option on the stack) or deselect it (i.e., this is a lasting decision). When an option that suspended its predecessor finishes execution, it automatically pops the stack and resumes the previous option. There are a few edge cases to handle (e.g., What to do if another option is selected while there are options on the stack?), but they're not difficult to manage. An example of the interfaces to support this can be found in our GAIA architecture [Dill 12c].

2.6 Knowledge Management

Knowledge is the key to good decision making. This is true in the real world, and it is doubly true in the realm of game AI, where most decisions boil down to relatively simple checks against the current situation. Of course, there are two different kinds of knowledge implied in that statement—knowledge of the situation itself, and knowledge of how to evaluate the situation in order to make a decision. Looked at that way, there isn't much to an AI beyond knowledge.

Given knowledge's central role in game AI, it's worth putting effort into thinking about how to best store and access our knowledge.

2.6.1 Blackboards

In the academic AI community, blackboard architectures typically refer to a specific approach in which multiple reasoners propose potential solutions (or partial solutions) to a problem, and then share that information on a *blackboard* [Wikipedia 12, Isla et al. 02]. Within the game community, however, the term is often used simply to refer to a shared memory space which various AI components can use to store knowledge that may be of use to more than one of them, or may be needed multiple times. In our architecture, for example, every character has access to two blackboards. The *character blackboard* stores